Lecture 11: Swapping, Memory Allocation, Memory Sharing

Yiying Zhang
Announcements

• Project 1 and midterm graded
• Midterm regraded with several issues corrected
  ♦ Sorry about mistakes on our part and thank you for those who pointed them out!

• Project 2 and homework 3 out

• Attend discussion section for project and homework!
Managing Page Tables

• How can we reduce page table space overhead?
  ♦ Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire addr space)

• How can we be flexible?
  “All computer science problems can be solved with an extra level of indirection.”

  two-level page tables
Two-Level Page Tables
Translation Look-aside Buffer (TLB)

Virtual address

VPN
offset

VPN
PFN
...
VPN
PFN
...
VPN
PFN
...

TLB

Miss

Real page table

Hit

PFN
offset

Physical address
Lecture Overview

We’ll cover more virtual memory topics:

• Optimizations
  ♦ Managing page tables (space)
  ♦ Efficient translations (TLBs) (time)
  ♦ Demand paged virtual memory (swapping) (space)

• Recap address translation

• Memory allocation

• Advanced Functionality
  ♦ Sharing memory
  ♦ Copy on Write
[lec9] Sharing main memory

- Simple multiprogramming – 4 drawbacks
  - Lack of protection
  - Cannot relocate dynamically
    - dynamic memory relocation: base&bound
  - Single segment per process
    - dynamic memory relocation: segmentation, paging

- Entire address space needs to fit in mem
  - More need for swapping
  - Need to swap whole, very expensive!
The last drawback

- So far we’ve separated the process’s view of memory from the OS’s view using a mapping mechanism
  - Each sees a different organization
  - Allows OS to shuffle processes around
  - Simplifies memory sharing
  - What is the essence of the mechanism that enables this?

- But, a user process had to be completely loaded into memory before it could run

  → Wasteful since a process only needs a small amount of its total memory at any time (reference locality!)
Virtual Memory

• Definition: *Virtual memory* permits a process to run with only some of its virtual address space loaded into physical memory

• Key idea: Virtual address space translated to either
  ♦ Physical memory (small, fast) or
  ♦ Disk (backing store), large but slow

• Deep thinking – what made above possible?

• Objective:
  ♦ To produce the illusion of memory as big as necessary
Virtual Memory

• “To produce the illusion of memory as big as necessary”
  ✷ Without suffering a huge slowdown of execution
  ✷ What makes this possible?
  ✷ *Principle of locality*
    » Knuth’s estimate of 90% of the time in 10% of the code
    » There is also significant locality in data references
Virtual Memory Implementation

- Virtual memory is typically implemented via demand paging
- **demand paging:**
  - Load memory pages (from storage) “on demand”
  - paging with swapping, e.g., physical pages are swapped in and out of memory
Demand Paging
(paging with swapping)

- If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

- How to detect this?
  - In software?
Demand Paging
(paging with swapping)

• If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?

• Hardware/software cooperate to make things work
  ♦ Extend PTEs with an extra bit “present” (valid bit)
  ♦ Any page not in main memory right now has the “present” bit cleared in its PTE
  ♦ If “present” isn’t set, a reference to the page results in a trap by the paging hardware, called page fault
  ♦ What needs to happen when page fault occurs?
x86 Page Table Entry

<table>
<thead>
<tr>
<th>Page frame number</th>
<th>U</th>
<th>P</th>
<th>Cw</th>
<th>Gl</th>
<th>L</th>
<th>D</th>
<th>A</th>
<th>Cd</th>
<th>Wt</th>
<th>O</th>
<th>W</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Valid (present)**
- **Read/write**
- **Owner (user/kernel)**
- **Write-through**
- **Cache disabled**
- **Accessed (referenced)**
- **Dirty**
- **PDE maps 4MB**
- **Global**

12 bits reserved
What is happening before and after malloc()? 

- Before malloc()? 
- After malloc()? 
- Upon first access? 

- How to capture the first write to a virtual page? 
  - e.g. want to trap into page fault handler 
    » Use valid bit 
  - In handler, check if vpage is allocated (and access permitted) 
    » If not, segmentation fault 
    » Else allocate physical page
Page Fault Handling in Demand Paging

- Load M
- Inst
- seq.
- mem
- ref
- Page Table (TLB)
- fault
- VM subsystem
- physical pages
Page fault handling (cont)

• On a page fault
  ♦ Find an unused phy. page or a used phy. page (how?)
  ♦ If the phy. page is used
    » If it has been modified (how to know?), write it to disk
    » Invalidate its current PTE and TLB entry (how?)
  ♦ Load the new page from disk
  ♦ Update the faulting PTE and its TLB entry
  ♦ Restart the faulting instruction

• Supporting data structure
  ♦ For speed: A list of unused physical pages (more later)
  ♦ Data structure to map a phy. page to its pid(s) and virtual address(es)
    » Sounds familiar?
We started this topic with the high-level problem of translating virtual addresses into physical addresses.

We’ve covered all of the pieces:
- Virtual and physical addresses
- Virtual pages and physical page frames
- Multi-level page tables and page table entries (PTEs)
- TLBs
- Demand paging

Now let’s put it together, bottom to top.
The Common Case

- Situation: Process is executing on the CPU, and it issues a read to an address
  - What kind of address is it, virtual or physical?
- The read goes to the TLB in the MMU
  1. TLB does a lookup using the page number of the address
  2. Common case is that the page number matches, returning the physical page frame and protection bits for this address
  3. TLB validates that the protection bits allows reads (in this example)
  4. MMU combines the PFN and offset into a physical address
  5. MMU then reads from that physical address, returns value to CPU
- Note: This is all done by the hardware
TLB Misses

- At this point, two other things can happen
  1. TLB does not have this virtual address
  2. Mapping in TLB, but memory access violates protection bits
- We’ll consider each in turn
Reloading the TLB

• If the TLB does not have mapping, two possibilities:
  1. MMU loads PTE from page table in memory
     » Hardware managed TLB, OS not involved in this step
  2. Trap to the OS
     » Software managed TLB, OS intervenes at this point
   ♦ A machine will only support one method or the other

• When TLB has PTE, it restarts translation
  ♦ Common case is that the PTE refers to a valid page in memory
    » Hardware just reads PTE from the page table and loads it into TLB
  ♦ Uncommon case is that TLB faults again on PTE because of PTE protection/valid bits (e.g., page is invalid (not in memory))
    » Becomes a page fault…
Page Faults

- PTE can indicate the type of a page fault
  - Read/write/execute – operation not permitted on page
  - Invalid – page not in physical memory

- TLB traps to the OS (software takes over)
  - R/W/E – OS usually will send fault back up to user process, or use for other purposes (e.g., copy on write, mapped files)
  - Invalid
    - Page not in physical memory because this is the first access
      - OS allocates physical frame and sets up the PTE (and flush TLB)
    - Page not in physical memory because it has been swapped out
      - Finds an empty frame in physical memory (if none, need to swap out something first), reads the page from disk, sets up the PTE to point to the new physical frame (and flush TLB)
Memory Allocation
Virtual memory allocation: two general forms

• Stack
  ♦ Restricted
  ♦ Simple and efficient
  ♦ Easy to implement

• Heap
  ♦ More general
  ♦ Less efficient
  ♦ More difficult to implement
Heap organization

- Allocation & freeing are unpredictable
  - For arbitrary, complex data structures

- Memory consists of allocated areas and free areas (holes) → lots of holes inevitable

- Fragmentation problem
  - solution: keep # of holes small, size large
Heap organization

- **Fragmentation**: inefficient use of memory due to holes too small
  - What happens in stack?

- Typically, heap allocation uses a *free list (or tree)* of holes
- Allocation algorithms differ in how to manage the free list
Implementation

• Bit map
  ♦ For fixed-size chunks

• Pools
  ♦ A separate allocation pool for each popular size
  ♦ Fast, no fragmentation
  ♦ But some pools may run out faster than others
Implementation – Segregated Lists

- Basic motivation: applications may use certain types of objects often
  - These types have fixed sizes

- Always keep a few free objects (memory regions) of popular sizes
  - One list of free objects for one size

- Example: Linux kernel slab allocator
Implementation - Buddy Allocation

• Basic idea: coalescing free regions
  ♦ Free space organized in “binary”
  ♦ Only give out power-of-two-sized blocks
  ♦ Neighboring free spaces form a bigger one

Internal Fragmentation!
Reclamation

• When can dynamically-allocated memory be freed?
  ♦ Easy if a chunk is used in one place
  ♦ Hard when a chunk is shared
  ♦ Sharing is indicated by presence of pointers to the data

• Reference counts:
  ♦ Keep track of the number of outstanding pointers to each chunk of memory
  ♦ When this goes to 0, free the memory
malloc, brk, and physical memory allocation

• Who calls malloc?
• What happens at malloc time?

• What is brk?
• Who calls brk?
• What happens at brk time?

• When is physical memory allocated?
malloc and \textit{brk} / \textit{mmap}

1. \texttt{malloc()}
2. \texttt{brk()}
3. \texttt{mmap()}
4. page fault

Application

Allocator (libc)

1. \texttt{malloc()}
2. \texttt{free()}
3. \texttt{realloc()}
4. \texttt{calloc()}

Virtual Memory

Heap

Mappings

Process Address Space

MMU

Physical Memory

lookup
Advanced Functionality

- Now we’re going to look at some advanced functionality that the OS can provide applications using virtual memory tricks
  - Shared memory
  - Copy on write
Sharing

- Private virtual address spaces protect applications from each other
  - Usually exactly what we want
- But this makes it difficult to share data (have to copy)
  - Parents and children in a forking Web server or proxy will want to share an in-memory cache without copying
- We can use shared memory to allow processes to share data using direct memory references
  - Both processes see updates to the shared memory segment
    - Process B can immediately read an update by process A
  - How are we going to coordinate access to shared data?
Sharing (2)

• How can we implement sharing using page tables?
  ♦ Have PTEs in both tables map to the same physical frame
  ♦ Each PTE can have different protection values
  ♦ Must update both PTEs when page becomes invalid

• How to destroy a virtual address space without affecting the other address space that shares data with it?
  ♦ Reference count

• How to swap out/in a shared page?
  ♦ Link all PTEs
  ♦ Operation on all entries
Isolation: No Sharing

Virtual Address Space #1 → Physical Memory → Virtual Address Space #2

...
Sharing Pages

Virtual Address Space #1

PTEs Point to Same Physical Page

Virtual Address Space #2
Copy on Write

- OSes spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()
- Use copy-on-write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying pages, create shared mappings of parent pages in child virtual address space
  - Shared pages are protected as read-only in parent and child
    » Reads happen as usual
    » Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction
- How does this help fork()?
Copy on Write: Before Fork

Parent Virtual Address Space

Physical Memory
Copy on Write: Fork

Parent Virtual Address Space

Physical Memory

Child Virtual Address Space

Read-Only Mappings
Copy on Write: On A Write

Parent Virtual Address Space

Physical Memory

Child Virtual Address Space

Now Read-Write & Private
Summary

Paging mechanisms:
• Optimizations
  ♦ Managing page tables (space)
  ♦ Efficient translations (TLBs) (time)
  ♦ Demand paged virtual memory (space)
• Recap address translation
• Memory allocation and reclamation
• Advanced Functionality
  ♦ Sharing memory
  ♦ Copy on Write

Next time: Paging policies
Next time...

- Chapter 22
Backup Slides
Mapped Files

- Mapped files enable processes to do file I/O using loads and stores
  - Instead of “open, read into buffer, operate on buffer, …”
- Bind a file to a virtual memory region (mmap() in Unix)
  - PTEs map virtual addresses to physical frames holding file data
  - Virtual address $base + N$ refers to offset $N$ in file
- Initially, all pages mapped to file are invalid
  - OS reads a page from file when invalid page is accessed
  - OS writes a page to file when evicted, or region unmapped
  - If page is not dirty (has not been written to), no write needed
    - Another use of the dirty bit in PTE
NAME

mmap, munmap - map or unmap files or devices into memory

SYNOPSIS

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
            int fd, off_t offset);
int munmap(void *addr, size_t length);
```

See NOTES for information on feature test macro requirements.

DESCRIPTION

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in `addr`. The `Length` argument specifies the length of the mapping (which must be greater than 0).

If `addr` is NULL, then the kernel chooses the (page-aligned) address at which to create the mapping; this is the most portable method of creating a new mapping. If `addr` is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping; see MAP_ANONYMOUS below), are initialized using `Length` bytes starting at offset `offset` in the file (or other object) referred to by the
MapViewOfFile function

Maps a view of a file mapping into the address space of a calling process.

To specify a suggested base address for the view, use the `MapViewOfFileEx` function. However, this practice is not recommended.

**Syntax**

```c++
LPVOID WINAPI MapViewOfFile(
    _In_ HANDLE hFileMappingObject,
    _In_ DWORD dwDesiredAccess,
    _In_ DWORD dwFileOffsetHigh,
    _In_ DWORD dwFileOffsetLow,
    _In_ SIZE_T dwNumberOfBytesToMap
);
```
Mapped Files

- Pages of file mapped one-to-one and contiguous into virtual pages in the address space
- Pages do not have to be contiguous in *physical* memory
- Not all pages have to be in physical memory at once
Mapped Files (2)

- File is essentially backing store for that region of the virtual address space (instead of using the swap file)
  - Virtual address space not backed by “real” files also called Anonymous VM

- Advantages
  - Uniform access for files and memory (just use pointers)
  - Less copying

- Drawbacks
  - Process has less control over data movement
    - OS handles faults transparently
  - Does not generalize to streamed I/O (pipes, sockets, etc.)